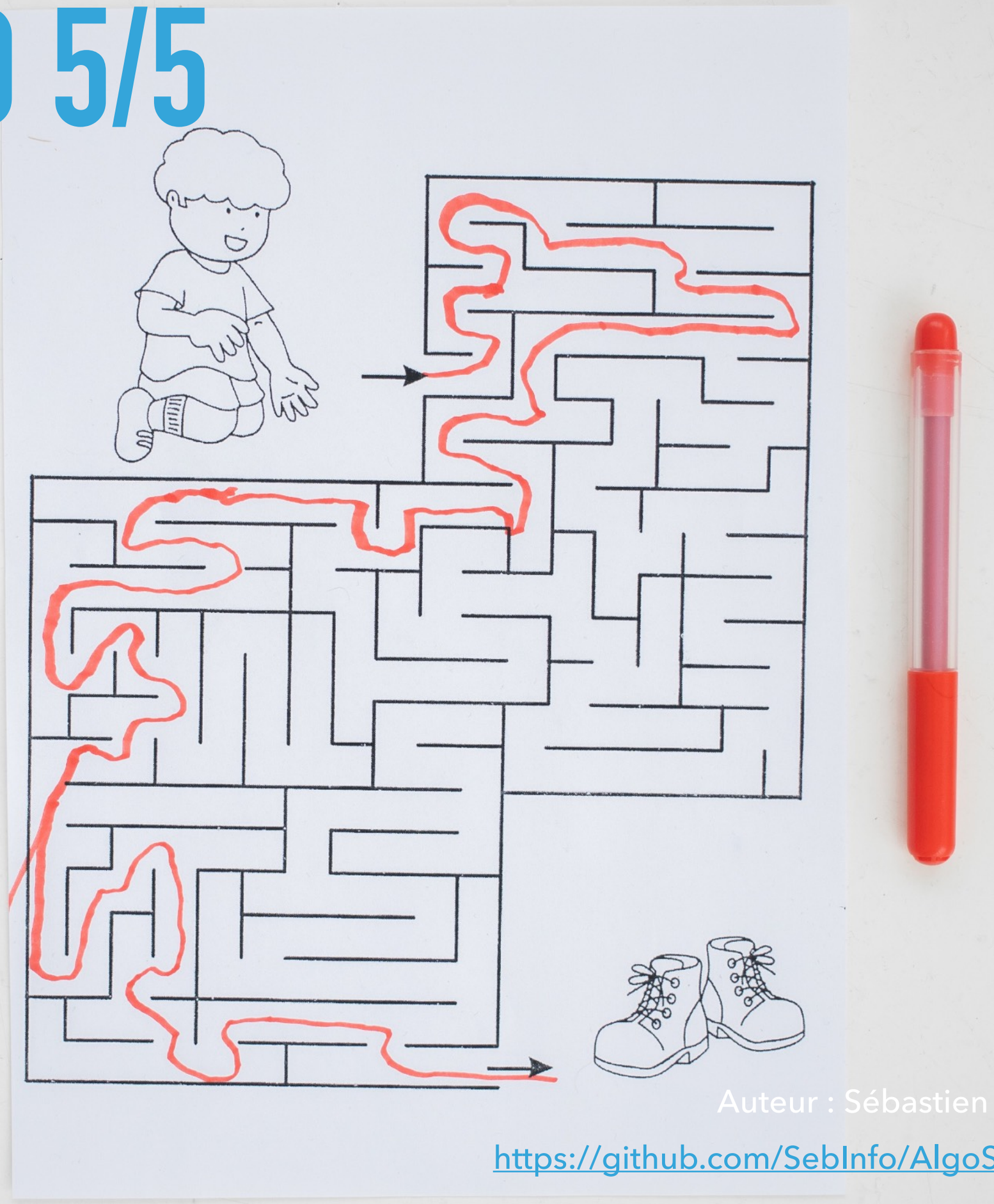
ALGO 5/5

studi



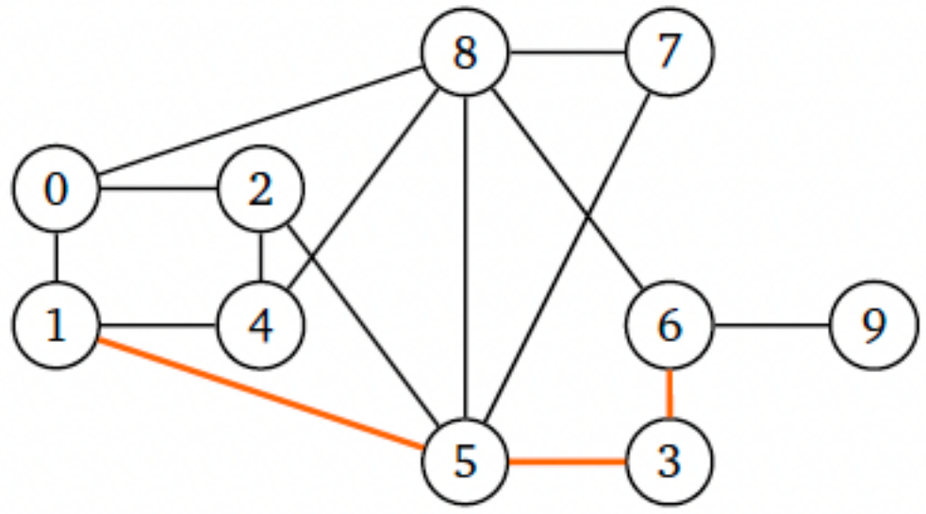
Auteur : Sébastien Inion

https://github.com/SebInfo/AlgoStudi5

GRAPHES

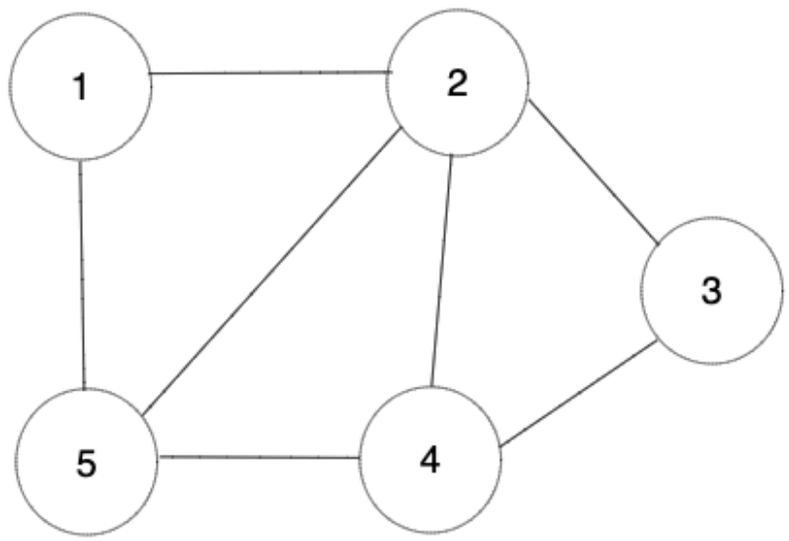
- Un graphe est un ensemble de sommets reliés entre eux par des arêtes.
- Deux sommets ne peuvent être reliés que par une arête au maximum.
- On appel degré d'un sommet son nombre de voisins

UN EXEMPLE DE GRAPHE



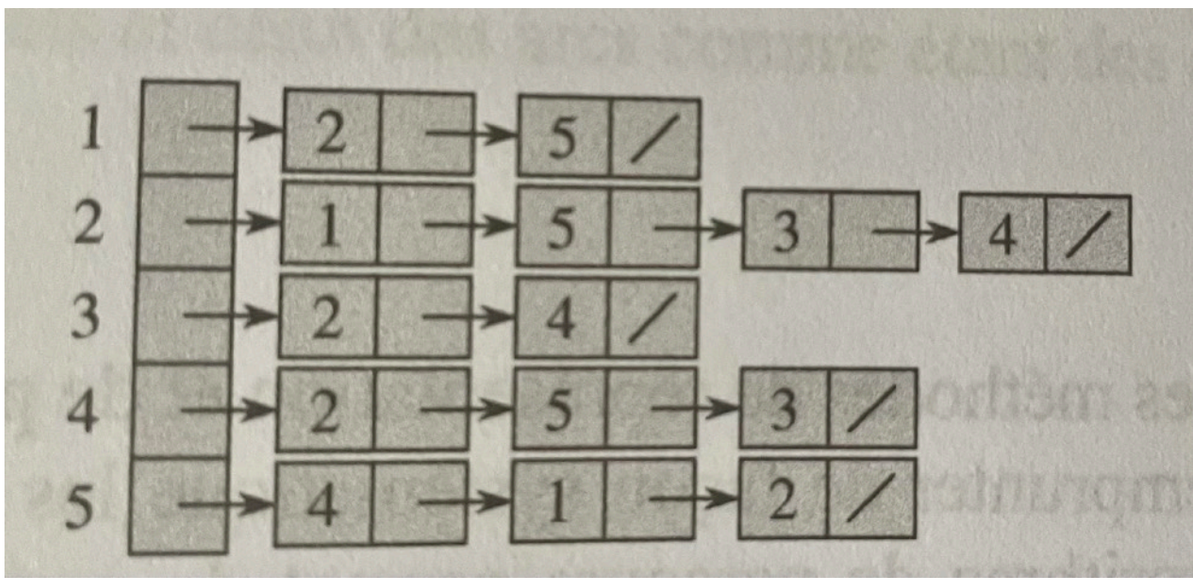
- Voici un graphe avec 10 sommets.
- Le sommet 8 est de degrés 5 (voisins 0,4,5,6,7).
- En rouge un chemin de longueur 3 allant du sommet 1 à 6.

REPRESENTATION DES GRAPHES

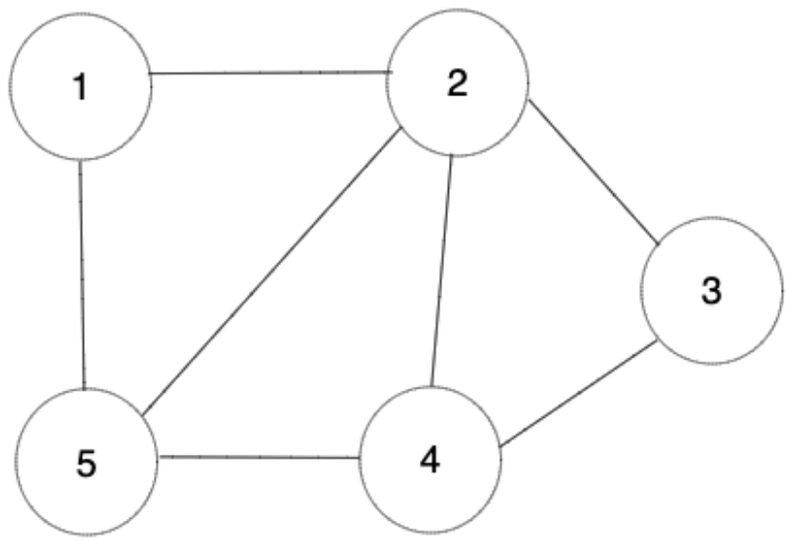


- On peut faire des listes d'adjacences

```
En Python
graph = [
    [2, 5],   #sommet 1
    [1, 5, 3, 4], # sommet 2
    [2, 4],
    [2, 5, 3],
    [4, 1, 2]
]
```

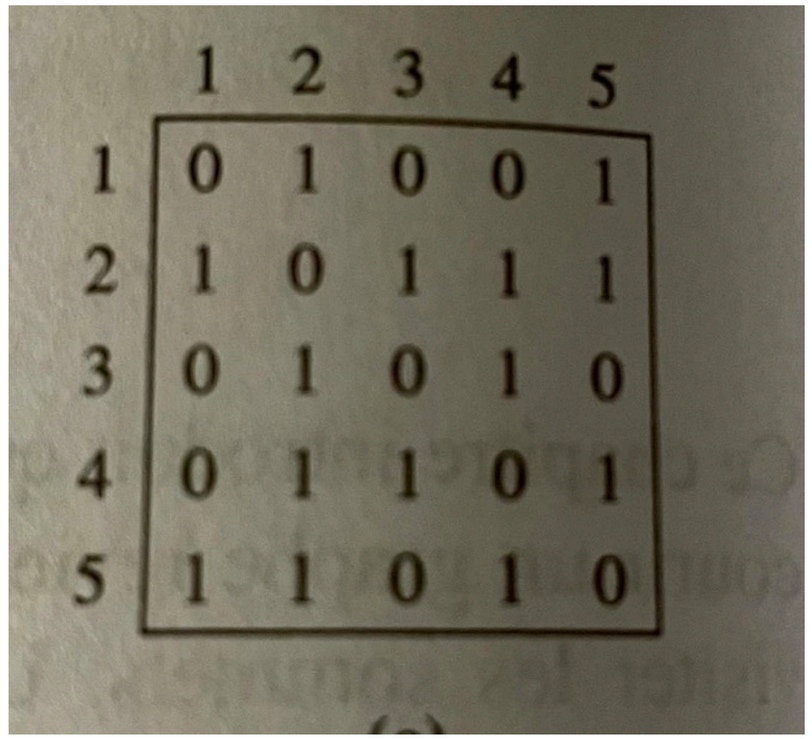


REPRESENTATION DES GRAPHES



- Ou des matrices d'adjacences

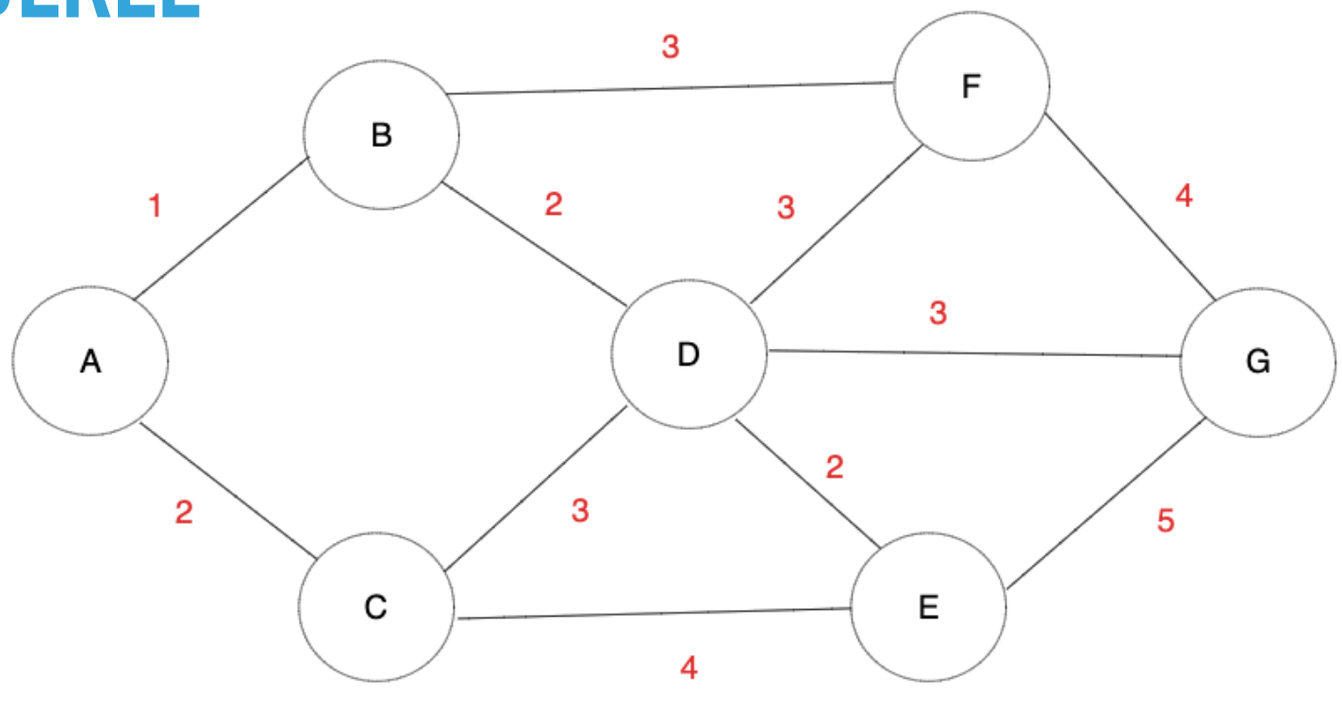
```
graph = [
    [0, 1, 0, 0, 1],  # Le sommet 1 est relié aux sommets 2 et 5
    [1, 0, 1, 1, 1],  # Le sommet 2 est relié aux sommets 1, 2, 3, 4, 5
    [0, 1, 0, 1, 0],  # Le sommet 3 est relié aux sommets 2, 4
    [0, 1, 1, 0, 1],  # Le sommet 4 est relié aux sommets 2, 3, 5
    [1, 1, 0, 1, 0]   # Le sommet 5 est relié aux sommets 1, 2, 4
]
```



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

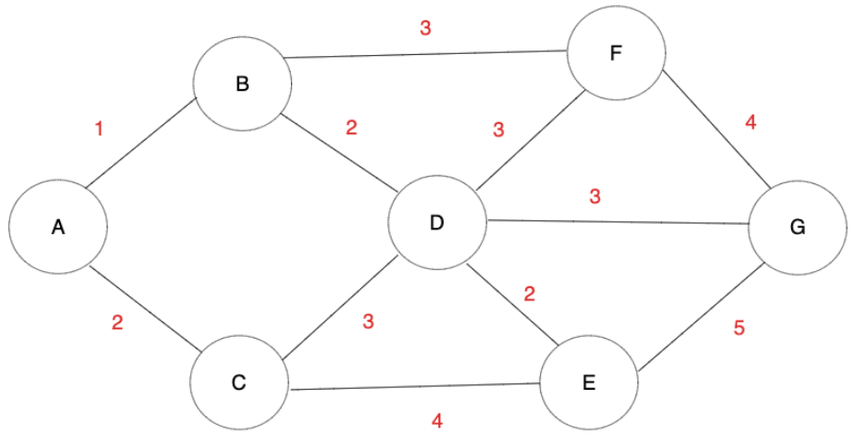
ALGO DE DIJKSTRA

GRAPHE PONDEREE



- Un graphe pondéré est un graphe dont chaque arête peut avoir une longueur différente.
- La longueur du chemin est alors définie comme la somme des longueurs des arêtes qui le compose.
- On peut facilement imaginer que les sommets sont des villes et les arêtes les chemins reliant ces villes.
- Par exemple entre le village A et le village B on a 1 km et entre le village D et F : 3 km
- Si l'on se pose comme problème de connaitre le plus court chemin entre la ville A et la ville G l'algorithme de Dijkstra offre la meilleur approche.

GRAPHE PONDEREE



A	B	C	D	E	F	G	Etapes
0	1.A	2.A					1
X	1.A		3.B		4.B		2
X	X	2.A	5.C	6.C			3
X	X	X	3.B	5.D	6.D	6.D	4
X	X	X	X		4.B	8.F	5
X	X	X	X	5.D	X	10.E	6
X	X	X	X	X	X	6.D	7

G->D->B->A (6km)

donc A->B->D->G

ALGO 5/5 - ALGO DE DIJKSTRA

DÉFINISSEZ LE GRAPHE :

- CHAQUE NŒUD REPRÉSENTE UN EMPLACEMENT DANS VOTRE ENVIRONNEMENT.
- CHAQUE ARÊTE REPRÉSENTE UNE CONNEXION ENTRE DEUX EMPLACEMENTS.
- CHAQUE ARÊTE EST ASSOCIÉE À UN POIDS, QUI REPRÉSENTE LA DISTANCE, LE COÛT OU LE TEMPS DE DÉPLACEMENT ENTRE LES EMPLACEMENTS.

INITIALISEZ LES STRUCTURES DE DONNÉES NÉCESSAIRES :

- CRÉEZ UN DICTIONNAIRE POUR STOCKER LES DISTANCES LES PLUS COURTES ENTRE LE POINT DE DÉPART ET LES AUTRES NŒUDS. INITIALISEZ TOUTES LES DISTANCES À L'INFINI, SAUF LA DISTANCE AU POINT DE DÉPART, QUE VOUS INITIALISEZ À 0.
- CRÉEZ UN DICTIONNAIRE POUR STOCKER LES NŒUDS PRÉCÉDENTS SUR LE CHEMIN LE PLUS COURT DEPUIS LE POINT DE DÉPART. INITIALISEZ-LES TOUS À NULL.
- CRÉEZ UN ENSEMBLE VIDE POUR SUIVRE LES NŒUDS VISITÉS.

DÉMARREZ L'ALGORITHME DE DIJKSTRA :

- TANT QUE VOUS AVEZ DES NŒUDS NON VISITÉS :
- SÉLECTIONNEZ LE NŒUD NON VISITÉ AVEC LA DISTANCE LA PLUS COURTE DEPUIS LE POINT DE DÉPART. CELA PEUT ÊTRE FAIT EN PARCOURANT LES NŒUDS NON VISITÉS ET EN SÉLECTIONNANT CELUI AVEC LA DISTANCE MINIMALE.
- MARQUEZ CE NŒUD COMME VISITÉ EN LE DÉPLAÇANT DE L'ENSEMBLE DES NŒUDS NON VISITÉS À L'ENSEMBLE DES NŒUDS VISITÉS.
- POUR CHAQUE NŒUD VOISIN DU NŒUD ACTUEL (NON VISITÉ) :
- CALCULEZ LA DISTANCE TEMPORAIRE À CE NŒUD VOISIN EN AJOUTANT LE POIDS DE L'ARÊTE ENTRE LE NŒUD ACTUEL ET LE NŒUD VOISIN À LA DISTANCE ACTUELLE DU NŒUD ACTUEL.
- SI LA DISTANCE TEMPORAIRE EST INFÉRIEURE À LA DISTANCE ENREGISTRÉE POUR LE NŒUD VOISIN, METTEZ À JOUR LA DISTANCE ENREGISTRÉE ET METTEZ LE NŒUD ACTUEL COMME LE PRÉCÉDENT NŒUD POUR LE NŒUD VOISIN.

RECONSTRUISEZ LE CHEMIN LE PLUS COURT :

- À LA FIN DE L'ALGORITHME, VOUS AVEZ LA DISTANCE MINIMALE ENTRE LE POINT DE DÉPART ET CHAQUE AUTRE NŒUD, AINSI QUE LE NŒUD PRÉCÉDENT SUR LE CHEMIN LE PLUS COURT DEPUIS LE POINT DE DÉPART.
- POUR TROUVER LE CHEMIN LE PLUS COURT ENTRE LE POINT DE DÉPART ET LE POINT D'ARRIVÉE, VOUS POUVEZ SUIVRE LES NŒUDS PRÉCÉDENTS À PARTIR DU POINT D'ARRIVÉE JUSQU'AU POINT DE DÉPART.

ALGO 5/5 - ALGO DE DIJKSTRA

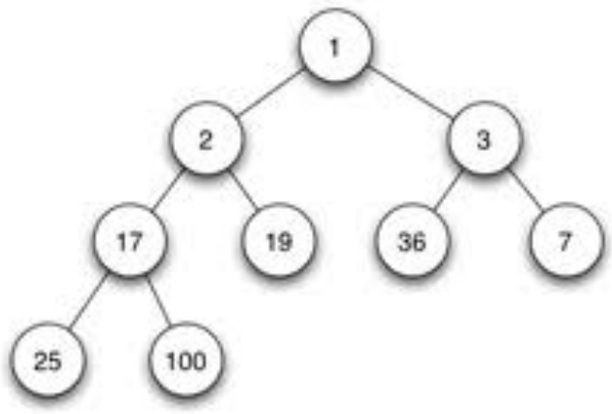
```
fonction Dijkstra(graphe, depart, arrivee):
    créer un ensemble vide nommé noeudsVisites
    créer un dictionnaire vide nommé distances
    créer un dictionnaire vide nommé predecesseurs

    initialiser distances[depart] à 0

    tant que arrivee n'est pas dans noeudsVisites:
        courant = le noeud avec la plus petite distance depuis depart qui n'est pas dans noeudsVisites

        ajouter courant à noeudsVisites

        pour chaque voisin de courant:
            si voisin n'est pas dans noeudsVisites:
                calculer distanceTentative = distances[courant] + poids(courant, voisin)

                si distanceTentative < distances[voisin]:
                    mettre à jour distances[voisin] avec distanceTentative
                    mettre à jour predecesseurs[voisin] avec courant

    // Reconstruction du plus court chemin
    chemin = liste vide
    courant = arrivee

    tant que courant n'est pas null:
        ajouter courant au début de chemin
        courant = predecesseurs[courant]

    retourner chemin
```

UTILISATION D'UN TAS BINAIRE

- Un tas binaire (binary heap) est une structure de données sous forme d'arbre binaire qui satisfait deux propriétés principales

- Propriété de tas (Heap property) : Pour un tas binaire min, chaque nœud (à l'exception du nœud racine) a une clé (valeur) supérieure ou égale à celle de son parent.

- En utilisant un tas binaire pour la file de priorité, l'algorithme de Dijkstra peut sélectionner efficacement le nœud non visité avec la plus petite distance à chaque itération, ce qui contribue à la complexité temporelle globale de l'algorithme.



ALGO 5/5 - ALGO DE DIJKSTRA

```python
def dijkstra(graph, start, end):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    previous_vertices = {vertex: None for vertex in graph}

    heap = [(0, start)]

    while heap:
        current_distance, current_vertex = heapq.heappop(heap)

        if current_vertex == end:
            break

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_vertices[neighbor] = current_vertex
                heapq.heappush(heap, (distance, neighbor))

    path = []
    current_vertex = end
    while current_vertex != start:
        path.insert(0, current_vertex)
        current_vertex = previous_vertices[current_vertex]
    path.insert(0, start)

    return path, distances[end]
```

COMPLEXITE DE L'ALGO DE DIJKSTRA

- Complexité d'ordre polynomiale
- Plus précisément, pour **n** sommets et **a** arcs, le temps est en $O((a+n)\log n)$
- Cela est dû au fait que chaque nœud est visité une fois n itérations, et pour chaque nœud, ses voisins sont examinés (a itérations au total) lors du calcul des distances.
- L'utilisation d'une structure de données telle qu'un tas binaire (binary heap) pour la file de priorité lors de la sélection du nœud avec la plus petite distance permet de maintenir une efficacité ***logarithmique***.

MERCI POUR VOTRE ATTENTION